Complexity - Introduction + Complexity classes

Computability - is a problem solvable?

Part I: via Turing machines

Part II: via Church's Lambda calculus

..now..

Part III: **Complexity** is concerned with how difficult a solvable problem is to solve..

its consumption of resources..

..not concerned with unsolvable problems.

In practice - if a problem cannot be solved in reasonable time it is no better than unsolvable to someone needing the solution.

We classify solvable problems into **Complexity Classes..**

P..the Class of tractable problems that can be solved efficiently (in polynomial time: p-time).

intractable problems are solvable but any algorithmic solution runs in exponential time (or slower) in the worst case. Practically unsolvable except for small inputs, unless average case much better than the worst.

NP..the class of problems which can be solved in p-time by a non-deterministic algorithm. Do they have deterministic p-time solutions? **"P = NP?"** if so, then all NP problems are in P..this has not been proved either way, but it is thought most likely that P ≠ NP, so problems in NP \P remain intractable (but not proved to be so).

NP-Complete problems..the hardest problems in NP. All NP-complete problems reduce to each other in p-time. Cook's theorem demonstrates that there are NP-complete problems (i.e. NP-complete is not an empty set)

Why do we study Complexity?..

• it guides us towards the tractable problems solvable with fast algorithms.

•..but we often encounter NP-complete problems in practice..so it will avoid (practically) hopeless searches for fast algorithms.

• the reducibility of every NP-complete problem to every other gives us a higher level view of solvability and the notion of algorithm and its formalism by TMs.

We will:

- define the run time function of a Turing machine
- introduce non-deterministic TMs and their run-time function
- formalise fast reduction of one problem to another
- examine NP and NP-complete problems

Some problems we will use in discussing complexity:

1. The Minimal Spanning Tree problem:

Given a connected weighted graph, G, find a spanning tree of the graph which has the shortest total weight.

2. The Hamiltonian Circuit problem:

Given a connected graph is there a circuit through the graph which visits each node exactly once. (the start/finish node counts once only)

3. The Travelling Salesman Problem

Given a complete weighted graph (there is an edge between every pair of nodes), and a value, d, is there a circuit which visits every node exactly once, with total path weight ≤ d?

4. The Propositional Satisfaction Problem (PSAT)

We write formulae of propositional logic, with **alphabet I** which includes:

atoms p1, p2, p3, ..

connectives $\wedge$ $\vee$ $\neg$ $\Rightarrow$ $\Leftrightarrow$ ( )

A formula is a word of I - can be input to a TM.

PSAT: given a formula A, is A satisfiable?

i.e. is there an assignment of true/false values to the atoms of A such that h(A) = true?

This has exponential run-time - it is an NP-complete problem.

Turing Machines for yes/no problems

Definition:

- a TM M is said to accept a word w of its input alphabet if
  M Halts and Succeeds on input w

- a TM M is said to reject a word w of its input alphabet if
  M Halts and Fails on input w

- M solves a yes/no problem A if
  - every instance of A is a word of M's input alphabet and
    - M accepts all yes-instances of A
    - M rejects no-instances of A

Examples of yes/no problems

Problem	instances	yes-instances	no-instances
is w prime?	binary rep. of numbers	bin. rep. of primes	bin. rep. of non-primes
Halting Problem (HP)	all pairs (code(M),w) M a standard TM, w a word of C	all (codeM),w) s.t. M H & S on input w	all (code(M),w) s.t. M does not H & S on input w
Hamiltonian Circuit Problem (HCP)	all finite graphs	graphs with a Hamiltonian circuit	graphs without a Hamiltonian circuit
Travelling Salesman Problem (TSP)	all pairs (G,d), G a weighted graph $d \geq 0$	pairs (G,d) s.t. G has a HC with length$\leq$d	pairs (G,d) s.t. G has no HC with length$\leq$d

We consider:

problems with infinitely many yes-instances

and infinitely many no-instances..

if finite .. could hard-wire them into the TM by just recognising whether the input was one of the finite number of (say) yes-instances ⇒ Halt and Succeed, and otherwise Halt and Fail..

….with no calculation being done by the TM.

yes ↑ yes-instances produce yes

or

no ↑ no-instances produce no

The result of running a TM to solve a yes/no problem:

Halt & Succeed: yes

Halt & Fail: no

We do not need output on the tape to get a result

The run-time function of a Turing Machine

$M = (Q, \sum, I, q_0, \delta, F)$

for input words w of length n (n=1, 2, 3..):

M runs a varying number of steps for various words w of length n.

define

$time_M$ (n) = length of longest run of M for input of length n

the function

$time_M (n) : \{0, 1, 2, ..\} \Rightarrow \{0, 1, 2, \ldots, \infty\}$

is the run-time function of M.

Polynomial-time (p-time) Turing Machines

”M runs in polynomial time”..means

“there is

$$p(n) = a_0+a_1n+a_2n^2+a_3n^3+..+a_kn^k$$

(a_0, a_1, a_2, a_3, ..a_k all non-negative integers)

such that

$$time_M(n) \leq p(n), \text{ all } n = 0,1,2,3..$$

Such a Turing Machine is **FAST**

p-time Turing Machines ALWAYS HALT.

Tractable problems

A yes/no problem is **tractable** if it can be solved by a TM running in p-time

intractable if it can be solved algorithmically, but not in p-time

An algorithm is **tractable** if it can be implemented by a p-time TM

intractable if it cannot be implemented by a p-time TM.

(the Cook-Karp thesis: “p-time TMs are fast”).

P is the class of tractable problems :

i.e. they can be implemented by a p-time TM.

The **complement of a problem in P**: exchange yes and no

eg. is n prime? ⟷ (complement) is n composite?

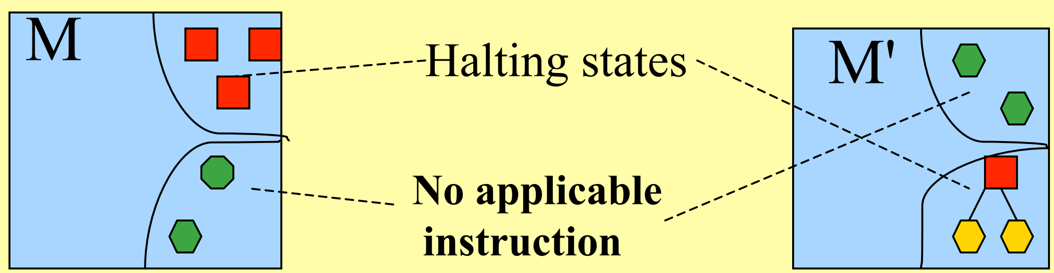
co-P: class of complements of problems in P.

P is closed under complementation:

if problem A ∈ P, then the complement of A ∈ P.

to complement A:

M solves problem A in p-time



Rewire M to Get M' solving complementary problem to A in p-time

P is closed under complementation

Intractable Problems

..some problems have been proved to be intractable
(cannot be solved by a p-time Turing Machine).

..many problems do not have a tractable algorithm, but have not yet been proven to be either tractable or intractable problems
..ie..is there a tractable algorithmic solution?..no proof yet,
but considered unlikely.
Examples: TSP, HCP, PSAT (propositional satisfaction problem).
..characterised by having a finite number of possible solutions
(but exponentially rising with the size of the problem instance)

..checking one to see whether it ***is*** a solution, may be done in p-time
..but in general
there is an exponential number of these checks to do.

Exhaustive search in algorithms

We have seen two main types of yes/no problems:

(∃) is there a solution?..solved if we can find one

we can check all possibilities, but this is intractable if the no. of possible solutions rises exponentially

or

(∀) is there NO solution? (complement of ∃ problem)

we can check all possibilities..intractable if the no. of possible solutions rises exponentially

Checking each possible solution can usually be done in p-time

eg. HCP, TSP.

Exhaustive search in algorithms..continued

A search strategy which homes in on a solution without exhaustive checking can render a problem tractable. eg. Minimum Spanning Tree The types ∃ and ∀ of problems each subdivide into those (∃1, ∀1) which have a search strategy, and those (∃2, ∀2) which have none

(∀) problems..eg. is every spanning tree of length > d?
..find a MST, calculate its weight, compare with d.
this is the complement of a tractable ∃ problem.

(∃2) and their complements (∀2) remain intractable..
is there a fast strategy to solve them?
if so, such a problem is in P ..
eg the minimal spanning tree problem
if not ..these are the NP-complete problems
eg. Hamiltonian Circuit, Travelling Salesman

Summary

We have introduced:

the time function of a Turing Machine

polynomial time function (p-time) TMs

Tractable and Intractable problems and algorithms

Complexity classes of problems

P ..can be solved by a deterministic TM in p-time

(for NP and NP-complete see later lectures).

C240 Computability & Complexity Coursework 1: Sample solution

The question asked for a 2-tape Turing machine; it is possible to use just one tape of a 2-tape TM, with the single read head moving between symbols of v and w, comparing them..similar to the "is w_1 equal to w_2?" TM in the notes, but with matching attempts starting in successive symbols of w, not just the first.

Most of the solutions submitted copied either v or w to the second tape, started the first matching attempt with the leftmost symbols of v and w,and moved the read heads right together along v and w matching symbols; when a mismatch is found:

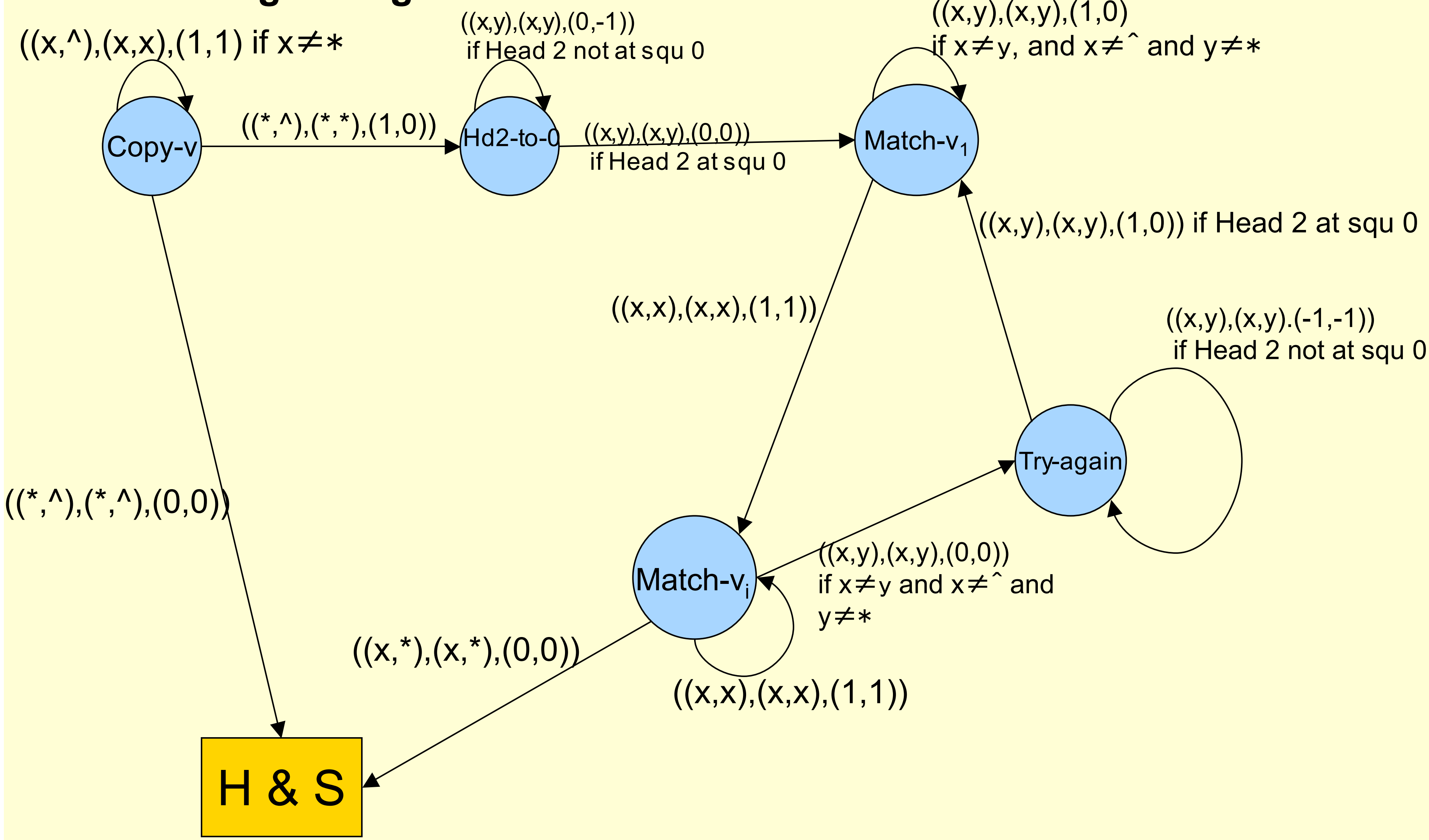
- if v has all been matched then H & S
- if the end of w has been reached without all of v matched, H & F
- else the heads are returned to the start of v and to one square in w after the start of the previous partial match, and symbol-by-symbol comparison of v with w restarts.

In the cases where v=ε ..H & S as ε is a substring of all strings
where w=ε; H & F unless v=ε too.

There were other variations:

- shifting w left 1 square after each unsuccessful attempt to match, so that both v and unmatched part of w started in square 0… this was used in some 1-tape solutions using 2 tracks.
- matching v starting with the last symbol of v and moving the heads together to the left; the first matching attempt either starting with the last symbol of w and successive attempts starting from 1 square further to the left each time, or starting with the leftmost possible match in the m^{th} symbol of w with successive attempts starting 1 square to the right.

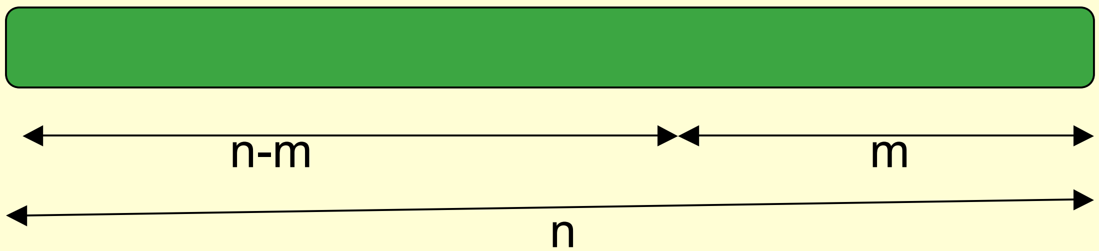
The Substring Turing machine



Time Function.

Worst case: for n> m, is where the first m-1 symbols of v match starting at each symbol of w:

Eg w = aaaa…aaa, v = aaab.



Number of steps:
Copy-v: m
Transition to Hd2-to-0: 1
Hd2-to-0: m
Transition to Match-v_1: 1 Initialisation total: 2m+2

For each of n-m+1 attempts to match v with w:

Match v_1 to Match-v_i:1
Transition to Try-again:1
Try-again: m-1
Transition to match-v_1: 1 total for matches within w: (n-m+1)x(2m+1)

Final match attempt (fails at end of w) (no rewind) 1+m-1
TOTAL: $2m+2 \; +2mn+n-2m^2 \; -m+2m+1 \quad +1+m-1 = 2mn-2m^2 +n+4m+3=2m(n-m+2) +(n+2)$